LEARN C++
Skill up with our free tutorials

2.11 — Header files

ALEX[1] JANUARY 11, 2024

Headers, and their purpose

As programs grow larger (and make use of more files), it becomes increasingly tedious to have to forward declare every function you want to use that is defined in a different file. Wouldn't it be nice if you could put all your forward declarations in one place and then import them when you need them?

C++ code files (with a .cpp extension) are not the only files commonly seen in C++ programs. The other type of file is called a **header file**. Header files usually have a .h extension, but you will occasionally see them with a .hpp extension or no extension at all. The primary purpose of a header file is to propagate declarations to code (.cpp) files.

Key insight

Header files allow us to put declarations in one location and then import them wherever we need them. This can save a lot of typing in multi-file programs.

Using standard library header files

Consider the following program:

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!";
    return 0;
}
```

This program prints "Hello, world!" to the console using *std::cout*. However, this program never provided a definition or declaration for *std::cout*, so how does the compiler know what *std::cout* is?

The answer is that *std::cout* has been forward declared in the "iostream" header file. When we `#include <iostream>`, we're requesting that the preprocessor copy all of the content (including forward declarations for std::cout) from the file named "iostream" into the file doing the #include.

Key insight

When you `#include` a file, the content of the included file is inserted at the point of inclusion. This provides a useful way to pull in declarations from another file.

Consider what would happen if the *iostream* header did not exist. Wherever you used *std::cout*, you would have to manually type or copy in all of the declarations related to *std::cout* into the top of each file that used *std::cout*! This would require a lot of knowledge about how *std::cout* was declared, and would be a ton of work. Even worse, if a function prototype was added or changed, we'd have to go manually update all of the forward declarations.

It's much easier to just `#include <iostream>`!

Using header files to propagate forward declarations

Now let's go back to the example we were discussing in a previous lesson. When we left off, we had two files, *add.cpp* and *main.cpp*, that looked like this:

add.cpp:

```
int add(int x, int y)
{
    return x + y;
}
```

main.cpp:

```
#include <iostream>

int add(int x, int y); // forward declaration using function prototype

int main()
{
    std::cout << "The sum of 3 and 4 is " << add(3, 4) << '\n';
    return 0;
}
```

(If you're recreating this example from scratch, don't forget to add *add.cpp* to your project so it gets compiled in).

In this example, we used a forward declaration so that the compiler will know what identifier *add* is when compiling *main.cpp*. As previously mentioned, manually adding forward declarations for every function you want to use that lives in another file can get tedious quickly.

Let's write a header file to relieve us of this burden. Writing a header file is surprisingly easy, as header files only consist of two parts:

1. A *header guard*, which we'll discuss in more detail in the next lesson (2.12 -- Header guards[2]).
2. The actual content of the header file, which should be the forward declarations for all of the identifiers we want other files to be able to see.

Adding a header file to a project works analogously to adding a source file (covered in lesson 2.8 -- Programs with multiple code files (https://www.learncpp.com/cpp-tutorial/programs-with-multiple-code-files/)[3]).

If using an IDE, go through the same steps and choose “Header” instead of “Source” when asked. The header file should appear as part of your project.

If using the command line, just create a new file in your favorite editor in the same directory as your source (.cpp) files. Unlike source files, header files should *not* be added to your compile command (they are implicitly included by #include statements and compiled as part of your source files).

Best practice

Prefer a .h suffix when naming your header files (unless your project already follows some other convention).

This is a longstanding convention for C++ header files, and most IDEs still default to .h over other options.

Header files are often paired with code files, with the header file providing forward declarations for the corresponding code file. Since our header file will contain a forward declaration for functions defined in *add.cpp*, we’ll call our new header file *add.h*.

Best practice

If a header file is paired with a code file (e.g. add.h with add.cpp), they should both have the same base name (add).

Here’s our completed header file:

add.h:

```
// 1) We really should have a header guard here, but will omit it for simplicity (we'll cover header guards in the next lesson)

// 2) This is the content of the .h file, which is where the declarations go
int add(int x, int y); // function prototype for add.h -- don't forget the semicolon!
```

In order to use this header file in main.cpp, we have to #include it (using quotes, not angle brackets).

main.cpp:

```cpp
#include "add.h" // Insert contents of add.h at this point.  Note use of double quotes here.
#include <iostream>

int main()
{
    std::cout << "The sum of 3 and 4 is " << add(3, 4) << '\n';
    return 0;
}
```

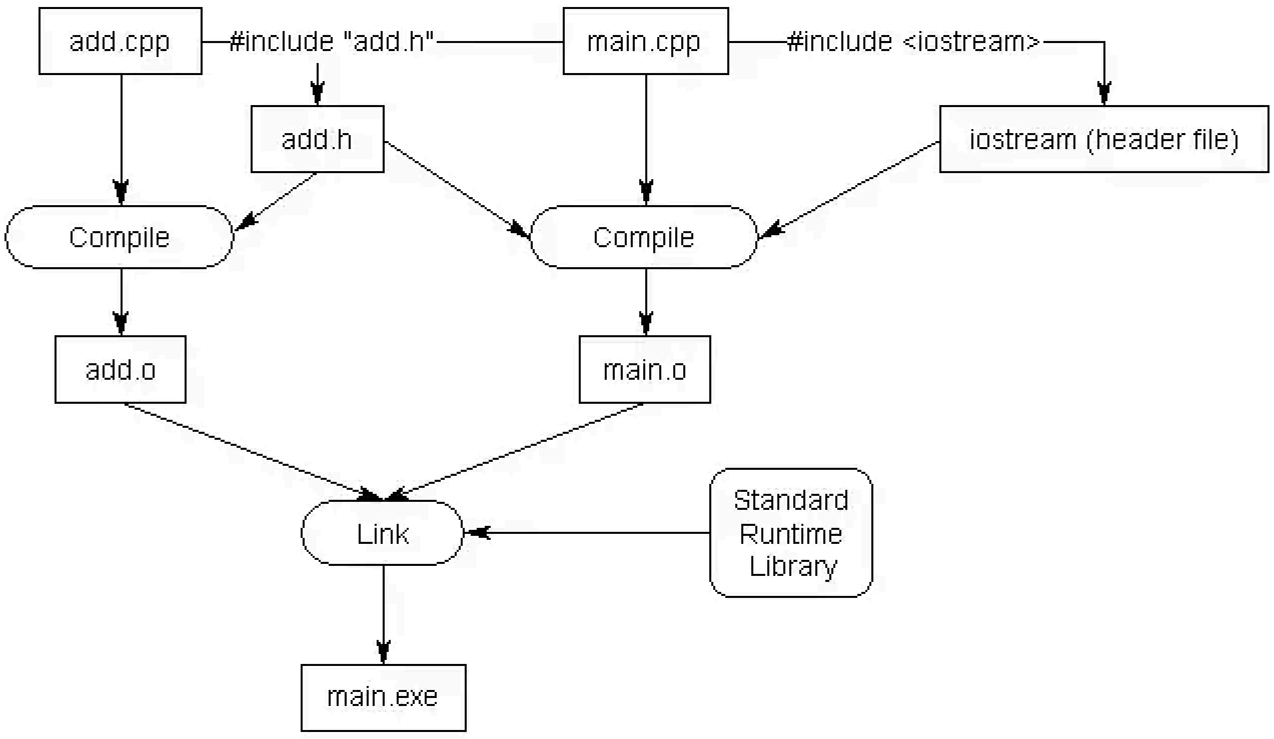
add.cpp:

```cpp
#include "add.h" // Insert contents of add.h at this point.  Note use of double quotes here.

int add(int x, int y)
{
    return x + y;
}
```

When the preprocessor processes the `#include "add.h"` line, it copies the contents of add.h into the current file at that point. Because our *add.h* contains a forward declaration for function *add()*, that forward declaration will be copied into *main.cpp*. The end result is a program that is functionally the same as the one where we manually added the forward declaration at the top of *main.cpp*.

Consequently, our program will compile and link correctly.



Note: In the graphic above, “Standard Runtime Library” should be labelled as the “C++ Standard Library”.

How including definitions in a header file results in a violation of the one-definition rule

For now, you should avoid putting function or variable definitions in header files. Doing so will generally result in a violation of the one-definition rule (ODR) in cases where the header file is included into more than one source file.

Related content

We covered the one-definition rule (ODR) in lesson 2.7 -- Forward declarations and definitions (https://www.learncpp.com/cpp-tutorial/forward-declarations/)[4].

Let's illustrate how this happens:

add.h:

```cpp
// We really should have a header guard here, but will omit it for simplicity (we'll cover header guards in the next lesson)

// definition for add() in header file -- don't do this!
int add(int x, int y)
{
    return x + y;
}
```

main.cpp:

```cpp
#include "add.h" // Contents of add.h copied here
#include <iostream>

int main()
{
    std::cout << "The sum of 3 and 4 is " << add(3, 4) << '\n';

    return 0;
}
```

add.cpp:

```cpp
#include "add.h" // Contents of add.h copied here
```

When `main.cpp` is compiled, the `#include "add.h"` will be replaced with the contents of `add.h` and then compiled. Therefore, the compiler will compile something that looks like this:

main.cpp (after preprocessing):

```cpp
// from add.h:
int add(int x, int y)
{
    return x + y;
}

// contents of iostream header here

int main()
{
    std::cout << "The sum of 3 and 4 is " << add(3, 4) << '\n';

    return 0;
}
```

This will compile just fine.

When the compiler compiles `add.cpp`, the `#include "add.h"` will be replaced with the contents of `add.h` and then compiled. Therefore, the compiler will compile something like this:

add.cpp (after preprocessing):

```cpp
int add(int x, int y)
{
    return x + y;
}
```

This will also compile just fine.

Finally, the linker will run. And the linker will see that there are now two definitions for function `add()`: one in main.cpp, and one in add.cpp. This is a violation of ODR part 2, which states, "Within a given program, a variable or normal function can only have one definition."

Best practice

Do not put function and variable definitions in your header files (for now).

Defining either of these in a header file will likely result in a violation of the one-definition rule (ODR) if that header is then #included into more than one source (.cpp) file.

Author's note

In future lessons, we will encounter additional kinds of definitions that can be safely defined in header files (because they are exempt from the ODR). This includes definitions for inline functions, inline variables, types, and templates. We'll discuss this further when we introduce each of these.

()Source files should include their paired header (#corresponding_include)[5]

In C++, it is a best practice for code files to #include their paired header file (if one exists). In the example above, *add.cpp* includes *add.h*.

This allows the compiler to catch certain kinds of errors at compile time instead of link time. For example:

something.h:

```
int something(int); // return type of forward declaration is int
```

something.cpp:

```
#include "something.h"

void something(int) // error: wrong return type
{
}
```

Because *something.cpp* #includes *something.h*, the compiler will notice that function *something()* has a mismatched return type and give us a compile error. If *something.cpp* did not #include *something.h*, we'd have to wait until the linker discovered the discrepancy, which wastes time. For another example, see this comment (https://www.learncpp.com/cpp-tutorial/header-files/comment-page-8/#comment-398571)[6].

We will also see many examples in future lessons where content required by the source file is defined in the paired header. In such cases, including the header is a necessity.

Best practice

Source files should #include their paired header file (if one exists).

()Do not #include .cpp files (#includecpp)[7]

Although the preprocessor will happily do so, you should generally not `#include` .cpp files. These should be added to your project and compiled.

There are number of reasons for this:

- Doing so can cause naming collisions between source files.
- In a large project it can be hard to avoid one definition rules (ODR) issues.
- Any change to such a .cpp file will cause both the .cpp file and any other .cpp file that includes it to recompile, which can take a long time. Headers tend to change less often than source files.
- It is non-conventional to do so.

Best practice

Avoid #including .cpp files.

Tip

If your project doesn't compile unless you #include .cpp files, that means those .cpp files are not being compiled as part of your project. Add them to your project or command line so they get compiled.

Troubleshooting

If you get a compiler error indicating that *add.h* isn't found, make sure the file is really named *add.h*. Depending on how you created and named it, it's possible the file could have been named something like *add* (no extension) or *add.h.txt* or *add.hpp*. Also make sure it's sitting in the same directory as the rest of your code files.

If you get a linker error about function *add* not being defined, make sure you've included *add.cpp* in your project so the definition for function *add* can be linked into the program.

()Angled brackets vs double quotes (#includemethod)[8]

You're probably curious why we use angled brackets for `iostream`, and double quotes for `add.h`. It's possible that a header file with the same filename might exist in multiple directories. Our use of angled brackets vs double quotes helps give the preprocessor a clue as to where it should look for header files.

When we use angled brackets, we're telling the preprocessor that this is a header file we didn't write ourselves. The preprocessor will search for the header only in the directories specified by the `include directories`. The `include directories` are configured as part of your project/IDE settings/compiler settings, and typically default to the directories containing the header files that come with your compiler and/or OS. The preprocessor will not search for the header file in your project's source code directory.

When we use double-quotes, we're telling the preprocessor that this is a header file that we wrote. The preprocessor will first search for the header file in the current directory. If it can't find a matching header there, it will then search the `include directories`.

Rule

Use double quotes to include header files that you've written or are expected to be found in the current directory. Use angled brackets to include headers that come with your compiler, OS, or third-party libraries you've installed elsewhere on your system.

Why doesn't iostream have a .h extension?

Another commonly asked question is "why doesn't iostream (or any of the other standard library header files) have a .h extension?". The answer is that *iostream.h* is a different header file than *iostream*! To explain requires a short history lesson.

When C++ was first created, all of the files in the standard library ended in a *.h* suffix. Life was consistent, and it was good. The original version of *cout* and *cin* were declared in *iostream.h*. When the language was standardized by the ANSI committee, they decided to move all of the names used in the standard library into the *std* namespace to help avoid naming conflicts with user-declared identifiers. However, this presented a problem: if they moved all the names into the *std* namespace, none of the old programs (that included iostream.h) would work anymore!

To work around this issue, a new set of header files was introduced that lack the *.h* extension. These new header files declare all names inside the *std* namespace. This way, older programs that include `#include <iostream.h>` do not need to be rewritten, and newer programs can `#include <iostream>`.

Key insight

The header files with the `.h` extension declare their names in the global namespace. They may also optionally declare those names in the `std` namespace as well.

The header files without the `.h` extension declare their names in the `std` namespace. Unfortunately and stupidly, these header files may optionally declare those names in the global namespace as well.

In addition, many of the libraries inherited from C that are still useful in C++ were given a *c* prefix (e.g. *stdlib.h* became *cstdlib*).

Best practice

When including a header file from the standard library, use the version without the .h extension if it exists. User-defined headers should still use a .h extension.

If a header file without a .h extension declares names into the global namespace, avoid those names, as they may not be available in the global namespace on other compilers. Prefer the names declared in the `std` namespace instead.

Including header files from other directories

Another common question involves how to include header files from other directories.

One (bad) way to do this is to include a relative path to the header file you want to include as part of the #include line. For example:

```
#include "headers/myHeader.h"
#include "../moreHeaders/myOtherHeader.h"
```

While this will compile (assuming the files exist in those relative directories), the downside of this approach is that it requires you to reflect your directory structure in your code. If you ever update your directory structure, your code won't work anymore.

A better method is to tell your compiler or IDE that you have a bunch of header files in some other location, so that it will look there when it can't find them in the current directory. This can generally be done by setting an *include path* or *search directory* in your IDE project settings.

For Visual Studio users

Right click on your project in the *Solution Explorer*, and choose *Properties*, then the *VC++ Directories* tab. From here, you will see a line called *Include Directories*. Add the directories you'd like the compiler to search for additional headers there.

For Code::Blocks users

In Code::Blocks, go to the *Project* menu and select *Build Options*, then the *Search directories* tab. Add the directories you'd like the compiler to search for additional headers there.

For GCC/G++ users

Using g++, you can use the -I option to specify an alternate include directory:

```
g++ -o main -I/source/includes main.cpp
```

There is no space after the -I.

For VS Code users

In your *tasks.json* configuration file, add a new line in the *"Args"* section:

```
"-I/source/includes",
```

The nice thing about this approach is that if you ever change your directory structure, you only have to change a single compiler or IDE setting instead of every code file.

()Headers may include other headers (#transitive)[9]

It's common that a header file will need a declaration or definition that lives in a different header file. Because of this, header files will often #include other header files.

When your code file #includes the first header file, you'll also get any other header files that the first header file includes (and any header files those include, and so on). These additional header files are sometimes called **transitive includes**, as they're included implicitly rather than explicitly.

The content of these transitive includes are available for use in your code file. However, you generally should not rely on the content of headers that are included transitively (unless reference documentation indicates that those transitive includes are required). The implementation of header files may change over time, or be different across different systems. If that happens, your code may only compile on certain systems, or may compile now but not in the future. This is easily avoided by explicitly including all of the header files the content of your code file requires.

Best practice

Each file should explicitly #include all of the header files it needs to compile. Do not rely on headers included transitively from other headers.

Unfortunately, there is no easy way to detect when your code file is accidentally relying on content of a header file that has been included by another header file.

Q: I didn't include <someheader> and my program worked anyway! Why?

This is one of the most commonly asked questions on this site. The answer is: it's likely working, because you included some other header (e.g. <iostream>), which itself included <someheader>. Although your program will compile, per the best practice above, you should not rely on this. What compiles for you might not compile on a friend's machine.

The #include order of header files

If your header files are written properly and #include everything they need, the order of inclusion shouldn't matter.

Now consider the following scenario: let's say header A needs declarations from header B, but forgets to include it. In our code file, if we include header B before header A, our code will still compile! This is because the compiler will compile all the declarations from B before it compiles the code from A that depends on those declarations.

However, if we include header A first, then the compiler will complain because the code from A will be compiled before the compiler has seen the declarations from B. This is actually preferable, because the error has been surfaced, and we can then fix it.

Best practice

To maximize the chance that missing includes will be flagged by compiler, order your #includes as follows:

1. The paired header file
2. Other headers from your project
3. 3rd party library headers
4. Standard library headers

The headers for each grouping should be sorted alphabetically (unless the documentation for a 3rd party library instructs you to do otherwise).

That way, if one of your user-defined headers is missing an #include for a 3rd party library or standard library header, it's more likely to cause a compile error so you can fix it.

Header file best practices

Here are a few more recommendations for creating and using header files.

- Always include header guards (we'll cover these next lesson).
- Do not define variables and functions in header files (for now).
- Give a header file the same name as the source file it's associated with (e.g. `grades.h` is paired with `grades.cpp`).
- Each header file should have a specific job, and be as independent as possible. For example, you might put all your declarations related to functionality A in A.h and all your declarations related to functionality B in B.h. That way if you only care about A later, you can just include A.h and not get any of the stuff related to B.
- Be mindful of which headers you need to explicitly include for the functionality that you are using in your code files, to avoid inadvertent transitive includes.
- A header file should #include any other headers containing functionality it needs. Such a header should compile successfully when #included into a .cpp file by itself.
- Only #include what you need (don't include everything just because you can).
- Do not #include .cpp files.
- Prefer putting documentation on what something does or how to use it in the header. It's more likely to be seen there. Documentation describing how something works should remain in the source files.

Next lesson
2.12 Header guards

2

Back to table of contents

10

Previous lesson
2.10 Introduction to the preprocessor

11

12

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[13] are connected to your provided email address.

996 COMMENTS Newest

MBA
April 22, 2024 10:50 am

Can we define one header file multiple times? i.e., can I use the std header (or any other) in the paired header file, and the source file as well ?

0 Reply

Alex Author
Reply to MBA [14] April 25, 2024 2:00 pm

I think you mean to ask if you can #include a given header file into multiple other files. If so, the answer is yes. Not only can you, but it's often a necessity.

1 Reply

MBA
Reply to Alex [15] April 27, 2024 2:22 am

Sorry for the confusing question. Thats not quite what i was trying to ask. Basically what I mean is, can I include any header that are defined in one header file in the same source file that the header is also defined in?

For example,
main.h includes #iostream header (or any other header, I'm only using iostream as I only know of this example).

main.cpp #includes main.h and also #includes iostream header.

Will that cause issues? Ofc you can spot this if the files are small but what if they're big and u include multiple headers that u also include in the source file.

0 Reply

groovy

Reply to MBA 14 April 25, 2024 10:27 am

if the .h file only contains function declarations then no dont include other libraries ex iostream

0 Reply

Baker

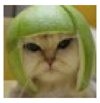
April 18, 2024 2:51 pm

Why would header files need declaration files from other header files? Aren't the actual functions defined elsewhere?

Headers forward declare functions, so the iostream header forward declares all the functions in the c++ standard library, but where is the standard library?

What is STD anyway? Isn't iostream the standard library? I thought STD was just a namespace. Are there multiple header files for iostream? Was that the solution that was made for the .h suffix?

0 Reply



Alex Author

Reply to Baker 16 April 20, 2024 11:13 am

For the same reason that .cpp file need declarations from other header files -- because those header files provide needed declarations.

The iostream header forward declares some I/O functions in the standard library, not all functions in the standard library.

The standard library is precompiled and distributed with your compiler. It gets linked in when the linker runs.

`std` is a namespace that contains declarations and definitions that belong to the standard library. This includes the content of iostream.

iostream includes other header files. You can see what it is required to include here: https://en.cppreference.com/w/cpp/header/iostream. Individual implementations may include other headers at their convenience.

2 Reply

tuta
April 2, 2024 1:12 pm

Related to **Including header files from other directories**. Consider a place holder header file in the project directory that utilizes a macro variable to refer to a relative path.

To illustrate this, let the **place holder header file** be `placeholder.h` and **relative path directory** be `C:\somepath\alex.h`. For the macro variable, I chose to suffix _HPC(Header Place Holder).

placeholder.h

```
#ifndef ALEX_HPC
#define ALEX_HPC "C:\somepath\alex.h" /*If the path is invalid a compiler
error will be flagged

                                       *The use of header guards is to
follow best practice. However, since this is the only
                                        explicit definition of ALEX_H, there
can be multiples copies (via multiple #includes)
                                        which wont result in an error as
redefined macros with identical values are
                                        allowed.

                                       */

#include ALEX_HPC                      //#include can be 'toggled off' by
commenting the line
#endif

//can be recursively adapted for multiple files. For example:

#ifndef NASCARDRIVER_HPC
#define NASCARDRIVER_HPC "C:\somepath\nascardriver.h"
#include NASCARDRIVER_HPC
#endif

#ifndef TUTA_HPC
#define TUTA_HPC "C:\somepath\tuta.h"
#include TUTA_HPC
#endif
```

main.cpp

```
#include placeholder.h //to include C:\somepath\alex.h
```

If one choses to include, specifically, a file that is not in the project directory, this approach solves the need to update all instances of `#include "C:\somepath\somename.h"` if the file path were to ever be changed. As opposed to using IDE project settings, this approach allows,

1. insightful comments to be made such as,

a. purpose of including a file, what it helps to achieve

```
#ifndef TUTA_HPC
#define TUTA_HPC "C:\somepath\tuta.h"
#include TUTA_HPC                           //can now generate random numbers
using rgenerate().
#endif
```

b. why yo u chose to include one header file over the other. (better optimized or the latter is outdated)

```
#ifndef ALEX_HPC
#define ALEX_HPC "C:\somepath\alex.h"
#include ALEX_HPC                           //bjarne.h is less optimized and
outdated
#endif
```

2. conditional compilation preprocessor directives to be included (outside the selected header file; in the place holder file).

0 Reply

tuta
Reply to tuta 17 April 2, 2024 1:15 pm

*I forgot to include header guards for the place holder header file.
Alex why is editing comments horrible?

0 Reply

Haven
March 25, 2024 1:00 pm

In the last best practice what do you mean by paired header file ?

And how can we know which "header file without a .h extension declares names into the global namespace" ?

0 Reply

Alex Author
Reply to Haven 18 March 25, 2024 2:44 pm

"Header files are often paired with code files, with the header file providing forward declarations for the corresponding code file."

> And how can we know which "header file without a .h extension declares names into the global namespace" ?

You can't (it's up to the compiler as to whether it does this or not), which is why you shouldn't rely on names being in the global namespace.

1 Reply

Brian
March 14, 2024 12:13 pm

I did not understand how to include header files from other directories. I use to command line and the command -I didnt work as it should.

0 Reply

Marco
Reply to Brian 19 March 26, 2024 11:59 pm

Learn to use CMakeList, much better than gcc command line only.

0 Reply

AVK
February 26, 2024 9:25 am

Hello Alex, you say :

Source files should include their paired header

In C++, it is a best practice for code files to #include their paired header file (if one exists)

Beyond the reasons for best practice, there is no other constraint (that the compiler or linker may enfore) that requires this correct ?

0 Reply

Alex Author
Reply to AVK 20 February 27, 2024 8:27 pm

Nope.

Obviously if the source file has a dependency on some declaration or definition in the header, then the source file will need to include the header to access that thing.

0 Reply

Appreciate
February 23, 2024 10:21 pm

What should I name my header file with multiple forward declarations, if it includes, substract() and add() foward declarations, or I should put add() and substract() forward declarations each in different header file?

If I changed certain function name, do I need to change the function name in all of the files manually, or there is an easier way to change all that function name all together automically?

Should the header file name be all small letters?

Last edited 2 months ago by Appreciate

0 Reply

Alex Author
Reply to Appreciate [21] February 25, 2024 8:19 pm

A single header should do. You could name it add.h (to match add.cpp) or calc.h or whatever you want.

If you change a function name you'll need to change all of the names manually or use global find/replace in your editor.

Header file names can be lower case or starting with a capital letter, depending on your preference. I often start with a capital letter when the header contains a program-defined type starting with a capital letter, and use lower-case otherwise.

1 Reply

IceFloe
February 21, 2024 1:15 am

thanks for the lesson, I want to ask you what to add.h do not have object files, but are simply included in the associated file add.cpp which is already compiled along with the header?

1 Reply

Alex Author
Reply to IceFloe [22] February 22, 2024 2:20 pm

Yes, the contents of add.h are added to each .cpp file that #includes add.h, and is compiled into the respective object file for that .cpp file.

0 Reply

Raju
February 18, 2024 8:50 pm

Please help getting error on visual studio
Cannot open include file: 'add.h': No such file or directory
cannot open source file "add.h"
identifier "add" is undefined
cannot open source file "add.h

0 Reply

Ajit Mali
January 28, 2024 2:43 am

As everyone know the VS code add file icon automatically depending upon the extension like the .c file has "blue C" icon, .cpp has "blue C++" icon, JavaScript has "Yellow JS" icon.

I give file extension to .h the logo was "purple C", fine.
but when I used .hpp or .hh the logo was "purple C++", if that's the case then for my understanding isn't .h used for c programs and .hpp used for cpp programs?

why it is better to use .h?

0 Reply

> **Alex** Author
> Reply to Ajit Mali [23] January 31, 2024 6:57 pm
>
> .h is the most conventional choice. But plenty of people prefer .hpp, and have valid reasons to do so. If you want to use .hpp (or .hh), it's fine.
>
> 1 Reply

Reuben
January 14, 2024 8:26 am

Hello, under the section **Using header files to propagate forward declarations**, why does the file `add.cpp` require the `#include "add.h"` directive? To my understanding, since the definition is a declaration as well, `add.cpp` should not require the include. Thanks!

1 Reply

> **LinuxCat**
> Reply to Reuben [24] January 17, 2024 10:48 am
>
> Read the "Source files should include their paired header" section.
>
> 0 Reply

molle
January 11, 2024 9:18 am

In the **Key insight** of the **Why doesn’t iostream have a .h extension?** section there must be one or a couple typos, because, as they are, the second half of these periods don't make sense grammatically, unless I'm missing something.

Relevant part of the text:

`The header files with the .h extension declared their names in the global namespace, and may optionally declared them in the std namespace as well.`
`The header files without the .h extension will declare their names in the std namespace. Unfortunately and stupidly, these header files may optionally declared those names in the global namespace as well.`

Last edited 3 months ago by molle

0 Reply

Alex Author
Reply to molle 25 January 11, 2024 9:28 pm

I updated the formatting of the insight box slightly to make the text easier to read. Better?

2 Reply

molle
Reply to Alex 26 January 12, 2024 6:35 am

Thank you for the swift intervention!
I'll make a donation soon, because I'm blocking ads and such good work shouldn't go unrewarded.
I especially appreciate that you're letting everyone access it for free. I believe that's the way of the future, at least for digital goods.
Cheers!

2 Reply

Gaurav Shah
December 28, 2023 6:05 pm

for VSCode users to add the header file path modify tasks.json arg section with "-I${workspaceFolder}\\nameOfHeaderFileDirectory" after "-g", argument

1 Reply

Daniil
December 20, 2023 11:03 pm

Why main.cpp(after preprocessing) has the include directive?

```
int add(int x, int y)
{
    return x + y;
}
include <iostream>

int main()
{
    std::cout << "The sum of 3 and 4 is " << add(3, 4) << '\n';

    return 0;
}
```

1 Reply

Alex Author
Reply to Daniil 27 December 22, 2023 12:44 pm

Removed the #include and replaced it with a comment indicating that's where the content of the iostream header would go. Thanks for pointing this out.

1 Reply

Sevgi
November 21, 2023 5:18 pm

Topics started to get harder, started taking notes. Thank you for this lesson

3 Reply

дмитрий
Reply to Sevgi 28 December 11, 2023 11:30 am

Тоже делаю, бещ этого никак уже

1 Reply

Junk Male
November 17, 2023 6:04 am

Is all this really necessary? I would've loved to get to the actual coding bit faster, the lessons are going too slow. Not meaning to attack, but it feels like we could've learned a lot of this info later on during practice. Sorry if I come across as rude, it's just getting hard keeping up with all this without getting distracted.

Last edited 5 months ago by Junk Male

2 Reply

Alex Author

Reply to Junk Male [29] November 17, 2023 5:03 pm

In a future revision of this website I'd like to move some of the declaration/definition/header stuff back a bit. But any such change will also necessite changes to any lesson with a dependency on this content, so it's not something to be done lightly.

5 Reply

Thanxxx

Reply to Alex [30] December 20, 2023 8:44 am

as a python programmer I appreciate this pace, but can definitely see the eagerness for newbies to get hands on more.

I think a suitable middle ground here could simply be making a simple calculator with main.cpp and operations.cpp operations.h add() sub() multi() div() .

edit: looks like its coming soon :D perhaps something smaller then. Anyway, thanks for the content.

Last edited 4 months ago by Thanxxx

1 Reply

cpp_coder

October 21, 2023 4:53 am

Was adding add.h to add.cpp necessary? Why?
-- should have read further :)

Last edited 6 months ago by cpp_coder

3 Reply

souhail

Reply to cpp_coder [31] March 10, 2024 5:24 am

because we prefer compiling errors rather than linking errors.

0 Reply

LBerserker
October 16, 2023 4:48 pm

So, "C++ Standard Library" is where "iostream" "lives", right?

0 Reply

Alex Author
Reply to LBerserker [32] October 18, 2023 12:02 pm

Yes.

3 Reply

LBerserker
Reply to Alex [33] October 21, 2023 6:20 am

Thank you!

1 Reply

Yopi
October 6, 2023 5:48 pm

Thanks for the lesson

7 Reply

OilSiren
September 30, 2023 10:15 am

When you say:
"This is because the compiler will compile all the declarations from B before it compiles the code from A that depends on those declarations."

Shouldn't it be the preprocessor?

0 Reply

Alex Author
Reply to OilSiren [34] October 2, 2023 10:11 am

No. The preprocessor just handles replacing the #includes with the included content. The compiler actually does the compiling of the included content.

1 Reply

Xavier
September 24, 2023 12:30 am

idk if you explicitly say so later, but when you say "Do not define variables and functions in header files." does this include things like enums/structs/classes and whatnot? or do you mean, "dont include int x{6};" Thank you! And loving these lessons so far, absolutely fantastic!

0 Reply

Alex Author
Reply to Xavier [35] September 26, 2023 7:18 pm

> does this include things like enums/structs/classes

It includes variables whose type is enum/struct/class, but not the type definitions themselves (which are okay to put in headers).

1 Reply

Xavier
Reply to Alex [36] September 26, 2023 8:42 pm

awesome, thank you!

1 Reply

snu
September 21, 2023 10:22 am

`Unlike source files, header files should not be added to your compile command (they are implicitly included by #include statements and compiled as part of your source files).`

I use the IDE CLion, which uses CMake. When I created the header file add.h, it automatically added the file to the add_executable command in the CMakeLists.txt:

```
add_executable(test main.cpp add.cpp add.h)
```

However since this tutorial states that header files don't need to be added to the compile command, I removed "add.h" and it still compiles and works fine. So I wonder why CLion automatically adds the header file? Is it necessary in some situations?

0 Reply

Alex Author
Reply to snu 37 September 22, 2023 11:17 am

Apparently this is how CLion keeps track of which headers are part of the project: https://stackoverflow.com/a/34716191

0 Reply

snu
Reply to Alex 38 September 22, 2023 11:20 am

Oh very interesting that CMake does not compile them even if they are mentioned in the command.
Thanks!

0 Reply

snu
September 21, 2023 10:11 am

Should you also use a main.h for the main.ccp?

1 Reply

Alex Author
Reply to snu 39 September 22, 2023 11:11 am

Only if you have anything in main.cpp that needs to be forward declared so it can be used in other .cpp files.

2 Reply

Eshan
September 17, 2023 10:37 am

if we use ${fileDirname}**.cpp in tasks.json will it compile the header files too?

0 Reply

Alex Author
Reply to Eshan 40 September 20, 2023 7:45 am

Header files are #included into .cpp files and compiled as part of those.

0 Reply

Eshan

Reply to Alex [41] September 20, 2023 11:25 am

oh right thanks

0 Reply

resident of flavourtown

September 3, 2023 4:10 pm

I have a question regarding header files:

For standard library or third party header files, made by yourself or others, when you `#include` a header, how does the definition get included? For header files that are part of the project, the accompanying .cpp file will get included by the linker, since it's part of the project, but for the headers outlined above, the .cpp file isn't already part of the project, so how will the linker know to link it?

0 Reply

Mikey

Reply to resident of flavourtown [42] January 1, 2024 5:33 pm

The definition gets included when the linker links in the C++ standard library which contains the definitions.

0 Reply

Alex Author

Reply to resident of flavourtown [42] September 5, 2023 1:26 pm

When you compile your project, each .cpp file is individually preprocessed and compiled. The preprocessor processes the #include and copies the content of the header into the .cpp at the point of inclusion. After preprocessing, the compiler compiles everything and produces an object file. After all files have been preprocessed and compiled, the set of object files is fed to the linker (along with any precompiled libraries) and the output target (usually an executable) is created.

0 Reply

Krishnakumar

August 25, 2023 12:13 pm

Brilliant lesson. Thanks Alex!

0 Reply

N T
August 10, 2023 12:26 am

I have a question regarding the section 'Do not #include .cpp files':

> Doing so can cause naming collisions between source files.
> In a large project it can be hard to avoid one definition rules (ODR) issues.

For the first reason, are you referring to the situation where a source file is included in another source file in addition to being compiled? Also, I'm not sure how these two reasons are distinct problems - won't naming collisions lead to ODR violations?

0 Reply

Alex Author
Reply to N T 43 August 12, 2023 2:15 pm

Not necessarily in addition to being compiled. These things can happen if a .cpp is included in another .cpp and only the first .cpp is compiled.

Naming collisions should lead to ODR violations, but ODR violations can occur even without a naming collision (e.g. if the same name is defined identically more than once).

1 Reply

Ashton Miller
July 23, 2023 8:12 pm

I am confused on a certain aspect of this whole thing. I wrote this code which has a main.cpp, multiplication.cpp, and a header.h. For some reason the multiplication.cpp will not work without its own #include <iostream>, but with the add.cpp example it did not need its own. why are these two situations different, and/or am I making a mistake elsewhere?

main.cpp

```cpp
#include <iostream>
#include "header.h"

int main()
    {
        std::cout << "enter any integer: "; //get first integer from user
        int x{}; //define integer one as x
        std::cin >> x;

        std::cout << "enter another integer: "; //get second integer from user
        int y{}; //define integer two as y
        std::cin >> y;

    multiplication(x, y); //initiate multiplication function

    return 0; //terminate program
    }
```

multiplication.cpp:

```cpp
int multiplication(int x, int y) //multiplication function
{
    std::cout << x << " multiplied by " << y << " is equal to " << x * y << '\n'; //multiply two integers and notify user

    return x * y; //return answer to main()
}
```

header.h:

```cpp
int multiplication(int x, int y); //declaring multiplication function
```

Last edited 9 months ago by Ashton Miller

0 Reply

vstar
Reply to Ashton Miller 44 August 7, 2023 8:22 pm

Oh , □□□iostream□□□

0 Reply

Alex Author
Reply to Ashton Miller 44 July 23, 2023 11:59 pm

Your multiplication.cpp has an output statement (using std::cout), so it requires iostream. My add.cpp doesn't do any output, so including iostream is not required.

1 Reply

Dan
July 22, 2023 3:17 pm

Undefined symbols for architecture arm64:
"add(int, int)", referenced from:
_main in main-4bb88d.o
ld: symbol(s) not found for architecture arm64
clang: error: linker command failed with exit code 1 (use -v to see invocation)

I'm using VSCode on Mac and I keep getting this link error. Been trying to resolve this for hours now. If anyone can help, it would be greatly appreciated!

0 Reply

Alex Author
Reply to Dan 45 July 23, 2023 10:26 pm

It sounds like add.cpp isn't being compiled into your program, so the linker can't find a definition for function `add(int, int)`.

0 Reply

Vad
July 19, 2023 8:30 am

Why does add.h need to be included into add.cpp? this basically means that we set the prototype and define function in one file, but why would we need to set prototype to add.cpp?

2 Reply

Alex Author
Reply to Vad 46 July 20, 2023 5:39 pm

It's explained in section `https://www.learncpp.com/cpp-tutorial/header-files/#corresponding_include`.

I also added a note that in future lessons we'll see cases where the source file requires content defined in the paired header. In such cases, including the paired header is required, not just a best practice.

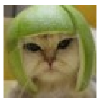
1 Reply

bigbawlsbobby69
July 11, 2023 3:36 pm

"**For now**, you should avoid putting function or variable definitions in header files... Do not put function and variable definitions in your header files (**for now**)."

Just noticed that 'for now' was mentioned here, referring to not putting functions/variables in header files. The explanation of how it can cause violations of the ODR makes sense, but in what situations would you realistically put a function or variable in a header file?

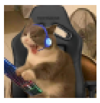
0 Reply


Alex Author
Reply to bigbawlsbobby69 47 July 13, 2023 12:41 pm

Later in this tutorial series, we discuss inline functions and inline variables, which can be put in header files.

There is a practical example of both here: https://www.learncpp.com/cpp-tutorial/generating-random-numbers-using-mersenne-twister/#RandomH

0 Reply


bigbawlsbobby69
Reply to Alex 48 July 13, 2023 12:58 pm

Ah, makes sense. Thank you :)

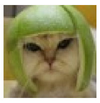
Last edited 9 months ago by bigbawlsbobby69

0 Reply

Anders
July 7, 2023 5:21 am

What is the best practice regarding capitalization of file names?
In the screenshots in 2.8 you named the file “Add.cpp” (with a capital A). Visual Studio auto-suggest the files names “Source.cpp” and “Header.h” respectively.
But everywhere in this chapter you refer to the files as “add.cpp”, “main.cpp”, “add.h” etc.
What do you recommend we use?

0 Reply


Alex Author
Reply to Anders 49 July 7, 2023 7:58 pm

I recommend you pick a style and be consistent (unlike me, apparently). :) There isn't a common convention here.

0 Reply

Ali
July 3, 2023 10:26 am

About why we should avoid including .cpp files, you said up there : "Any change to a .cpp file will cause the whole project to recompile".
Do you mean actually that if modify the source file and I include the header file, it will not cause the project to recompile ?

0 Reply

Alex Author
Reply to Ali 50 July 4, 2023 5:27 pm

I was inadvertently being a bit hyperbolic -- if you modify a .cpp file, that .cpp will need to recompile, as well as any other .cpp files that include it. .cpp files tend to change more often than header files, so if you include .cpp files you'll be recompiling larger parts of your project more often. I updated the text to reflect this.

2 Reply

Harsha Varthan
July 1, 2023 4:45 am

Can you please explain me how #include .cpp file can cause naming collisions between source files? Also, does naming collition and multipile definition error means the same thing?

0 Reply

Alex Author
Reply to Harsha Varthan 51 July 3, 2023 3:18 pm

If file a.cpp defines some variable named `a`, and file b.cpp defines some variable named `a` if file a.cpp #includes b.cpp then you will have two definitions for `a` in the same translation unit, and that will cause a compile error.

A naming collision occurs when we have two different entities that use the same name in the same scope, and the compiler can't disambiguate them. Multiple definitions occur when there is more than one definition for a given identifier in a translation unit.

For example:

```
int a{};
int a{};
```

The difference between the two is that in a naming collision case, the name is meant to be used for different things, whereas in the multiple definition case the definitions could be for the same object.

1 Reply

Harsha Varthan

Reply to Alex 52 July 10, 2023 5:27 am

Let us suppose there are two source files

Main.cpp

```
#include <iostream>

int add = 123;

int main(){
    std::cout << "Hello World" << '\n';
    return 0;
}
```

Math.cpp

```
int add(int a, int b){
    return a+b;
}
```

will this program thorw an error? if not why?

0 Reply

Alex Author

Reply to Harsha Varthan 53 July 12, 2023 11:15 pm

No. Each file compiles just fine and there are no uses of add that are ambiguous.

0 Reply

Simon Gancar.

June 24, 2023 2:07 pm

"When it comes to functions and variables, it's worth keeping in mind that header files typically only contain function and variable declarations, not function and variable definitions (otherwise a violation of the one definition rule could result)".

Why's that the case? I understand that reasoning for classes and templates as those can have diffrent definitions in diffrent files, but when it comes to my interpretation of the one definition rule, all functions and variablesare defined in another file and that definition is set in place for the rest of the program files, no matter if they are declared in that file or the header itself. Mind explaining what you meant? I'm curious and also wanna understand why I should put my definitions in a diffrent file then the declerations.

0 Reply

Alex Author

Reply to Simon Gancar. 54 June 24, 2023 10:36 pm

Added a section into this lesson illustrating this (How including definitions in a header file results in a violation of the one-definition rule)

0 Reply

Harsha Varthan

June 23, 2023 10:18 am

is it true that iostream header file just contains forward declaration on std::cout, and the actual definition is defined in iosfwd header file?

Last edited 10 months ago by Harsha Varthan

1 Reply

Alex Author

Reply to Harsha Varthan 55 June 24, 2023 12:02 pm

iosfwd contains forward declarations (hence the name) of various types. iostream contains the forward declaration for std::cout.

The actual definition for std::cout is likely in an implementation defined .cpp file that is complied into your standard library runtime.

1 Reply

Ilya Chalov

June 18, 2023 4:35 am

> If using the command line, just create a new file in your favorite editor.

I think it makes sense to additionally say that header files do not need to be included in the command line, since their contents are included by the preprocessor in the code file.

(At first I thought that header files should also be included in the command line, since when working with the IDE they are included in the list of files in the solution explorer.)

I have done experiments with MSVC and g++ (GCC) compilers (the program consists of three files `main.cpp`, `add.cpp`, `add.h`):

Compiles without errors and the resulting executable file works correctly:

```
cl /EHsc /utf-8 /std:c++20 /W4 "main.cpp" "add.cpp" /link
/out:"program.exe"
```

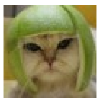
fatal error LNK1107:
`cl /EHsc /utf-8 /std:c++20 /W4 "main.cpp" "add.cpp" "add.h" /link /out:"program.exe"`

Compiles without errors in both cases and the resulting executable file works correctly:

`g++ "main.cpp" "add.cpp" "add.h" -o "program" -std=c++20 -Wall`

`g++ "main.cpp" "add.cpp" -o "program" -std=c++20 -Wall`

0 Reply



Alex Author
Reply to Ilya Chalov 56 June 19, 2023 1:04 pm

Good feedback. Lesson amended. Thank you!

0 Reply

Ilya Chalov
Reply to Alex 57 June 22, 2023 10:40 am

Cool!

> to your commmand line compile command

You have a typo there.

0 Reply

coderman
June 11, 2023 5:27 am

I have written a program that computes sum, difference, product and division of 2 given numbers and as such i have created 4 different files containing functions for each operation.

By doing forward declaration in the `main.cpp` file, the program works fine, but when I tried to do the same thing by making a header file `arithmetic.h` I am getting error that neither of the 4 functions are defined.

I have checked the spelling and stuff and they are right

main.cpp

```cpp
#include "arithmetic.h"
#include <iostream>

int main()
{
    std::cout << "Enter a number: ";
    float x {};
    std::cin >> x;

    std::cout << "Enter another number: ";
    float y {};
    std::cin >> y;

    std::cout << x << " + " << y << " = " << add(x, y) << "\n" ;
    std::cout << x << " - " << y << " = " << subtract(x, y) << "\n";
    std::cout << x << " * " << y << " = " << multiply(x, y) << "\n";
    std::cout << x << " / " << y << " = " << divide(x, y) << "\n";

    return 0;
}
```

add.cpp

```cpp
#include "arithmetic.h"

float add(float x, float y)
{
    return x + y;
}
```

subtract.cpp

```cpp
#include "arithmetic.h"

float subtract(float x, float y)
{
    return x - y;
}
```

multiply.cpp

```cpp
#include "arithmetic.h"

float multiply(float x, float y)
{
    return x * y;
}
```

divide.cpp

```cpp
#include "arithmetic.h"

float divide(float x, float y)
{
    return x / y;
}
```

arithmetic.h

```cpp
#ifndef ARITHMETIC_H_INCLUDED
#define ARITHMETIC_H_INCLUDED

float add(float x, float y);
float subtract(float x, float y);
float multiply(float x, float y);
float divide(float x, float y);

#endif // ARITHMETIC_H_INCLUDED
```

What is the problem here?

0 Reply

Alex Author

Reply to coderman 58 June 14, 2023 9:43 am

No idea. Works fine for me. Can you paste the exact error you are getting?

0 Reply

coderman

Reply to Alex 59 June 15, 2023 1:27 am

I am getting these errors in the `main.cpp` file:

```
|14| undefined reference to 'add(float, float)'|
|15| undefined reference to 'subtract(float, float)'|
|16| undefined reference to 'multiply(float, float)'|
|17| undefined reference to 'divide(float, float)'|
||error: ld returned 1 exit status|
||=== Build failed: 5 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===|
```

I think it somehow is not able to read the functions but know they exist because it knows that the arguments are of `float` type

0 Reply

Alex Author

Reply to coderman 60 June 18, 2023 8:04 pm ld is the linker, so these are linker errors. That would indicate that you are not properly compiling add.cpp, subtract.cpp, etc... into your program.

0 Reply

coderman

Reply to Alex 61 June 21, 2023 6:49 am

Whenever I try to compile those files I get another error:

`undefined reference to 'WinMain'`

Upon searching about it I got to know that this error is due to the absence of main() function. But there is no need for a main() function here. So what do I do?

Also in another project where I did the same thing but without using the header file (with forward declarations), there are no errors and even the add.cpp, etc files get compiled without any such error

0 Reply

Alex Author

Reply to coderman 62 June 23, 2023 9:16 pm

That WinMain error means you set up your project as a windows application rather than a console application. Recreate your project as a console application and try again.

0 Reply

Krishnakumar

May 26, 2023 7:18 am

```
> int something(int); // return type of forward declaration is int
```

This violates the (admittedly questionable?) best practice recommendation of including the identifiers of the parameter arguments, given in an earlier lesson.

```
#include "something.h"

void something(int) // error: wrong return type
{
}
```

Well, the above won't even compile even if the return type is fixed, because the function definition is missing parameter name.

0 Reply

Alex Author

Reply to Krishnakumar 63 May 30, 2023 12:04 am

Hmmm, the parameter in the function definition has no name, so technically the parameter in the forward declaration does have the same name. :)

A function definition is *not* required to have a parameter name. If you have a named parameter that you don't use inside the function, most compiles will warn you about an unused parameter. However, they won't warn for unnamed parameters (because there is no way to access them!).

It's common to leave a parameter that must exist but isn't used unnamed (both so the compiler doesn't generate a warning, and as an indication that the parameter isn't actually used for anything).

0 Reply

Krishnakumar

Reply to Alex 64 June 5, 2023 10:16 am

>A function definition is not required to have a parameter name.

Hi Alex, that is interesting. I didn't know about this.

> However, they won't warn for unnamed parameters (because there is no way to access them!).

Compiler doesn't, but static analysers do. Apparently this is frowned upon for readability aspects.

>It's common to leave a parameter that must exist but isn't used unnamed (both so the compiler doesn't generate a warning, and as an indication that the parameter isn't actually used for anything).

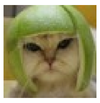
Apparently, this is considered poor practice at least by `clang-tidy`, which flags it under the categories `[hicpp-named-parameter,readability-named-parameter]` and is based on Google's C++ style guide https://google.github.io/styleguide/cppguide.html#Function_Declarations_and_Definitions. Here is the link to clang-tidy's explanation page for this: https://clang.llvm.org/extra/clang-tidy/checks/readability/named-parameter.html

The recommended way is to use a commented parameter specifically called `unused` like so:

```
returnType functionName(parameterType /*unused*/)
```

Last edited 10 months ago by Krishnakumar

0 Reply



Alex Author

Reply to Krishnakumar 65 June 5, 2023 5:41 pm

Added a section about this to lesson https://www.learncpp.com/cpp-tutorial/introduction-to-function-parameters-and-arguments/

It looks like clang-tidy is simply enforcing the Google recommendation.

The C++ Core Guidelines recommend omitting the name of the parameter: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-unused

0 Reply

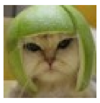
Krishnakumar

Reply to Alex 66 October 10, 2023 1:11 pm

>Added a section about this to lesson https://www.learncpp.com/cpp-tutorial/introduction-to-function-parameters-and-arguments/

Yes. This is perfect. Thanks, Alex.

0 Reply



Alex Author

Reply to Krishnakumar 65 June 5, 2023 5:09 pm

Interesting. Does clang-tidy flag `returnType functionName(parameterType /*unused*/)` ?

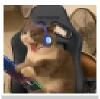
0 Reply

Krishnakumar

Reply to Alex 67 June 5, 2023 5:10 pm

`clang-tidy` actually recommends it.

0 Reply



bigbawlsbobby69

May 22, 2023 10:29 pm

Why do you use the `.h` extension over the `.hpp` extension? Are there certain advantages or disadvantages to using one over the other?

Thanks for any help! :)

0 Reply

Alex Author

Reply to bigbawlsbobby69 68 May 24, 2023 11:46 am

It's mainly stylistic. Most IDEs still use .h by default, and .h is a longstanding convention, so we recommend that.

1 Reply

ChatGPT

Reply to bigbawlsbobby69 68 May 23, 2023 2:29 am

The use of `.h` or `.hpp` file extensions for C++ header files is a matter of convention and personal preference. Both extensions are commonly used and understood by C++ developers, and there is no inherent advantage or disadvantage to using one over the other in terms of language features or functionality.

Traditionally, the `.h` extension has been used for C and C++ header files. It has become a widely recognized convention, and many existing codebases and libraries use this extension. The `.h` extension is often associated with C++ header files that contain only declarations or preprocessor directives.

On the other hand, the `.hpp` extension is sometimes used to differentiate C++ header files from C header files. The use of `.hpp` may imply that the header file contains C++ specific code or features, although it is not a strict rule. Some developers prefer to use `.hpp` to indicate that the file uses the C++ language.

Ultimately, the choice between `.h` and `.hpp` is subjective and depends on personal preference or project conventions. The most important aspect is to be consistent within a project or codebase to maintain readability and avoid confusion for other developers.

1 Reply

bigbawlsbobby69

Reply to ChatGPT 69 May 23, 2023 10:26 pm

Ahh, that makes sense. Thanks! :)

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/header-guards/
3. https://www.learncpp.com/cpp-tutorial/programs-with-multiple-code-files/
4. https://www.learncpp.com/cpp-tutorial/forward-declarations/
5. https://www.learncpp.com/cpp-tutorial/header-files/#corresponding_include
6. https://www.learncpp.com/cpp-tutorial/header-files/comment-page-8/#comment-398571
7. https://www.learncpp.com/cpp-tutorial/header-files/#includecpp
8. https://www.learncpp.com/cpp-tutorial/header-files/#includemethod
9. https://www.learncpp.com/cpp-tutorial/header-files/#transitive
10. https://www.learncpp.com/
11. https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/
12. https://www.learncpp.com/header-files/
13. https://gravatar.com/
14. https://www.learncpp.com/cpp-tutorial/header-files/#comment-596096
15. https://www.learncpp.com/cpp-tutorial/header-files/#comment-596202
16. https://www.learncpp.com/cpp-tutorial/header-files/#comment-595943
17. https://www.learncpp.com/cpp-tutorial/header-files/#comment-595363
18. https://www.learncpp.com/cpp-tutorial/header-files/#comment-595102
19. https://www.learncpp.com/cpp-tutorial/header-files/#comment-594693
20. https://www.learncpp.com/cpp-tutorial/header-files/#comment-594057
21. https://www.learncpp.com/cpp-tutorial/header-files/#comment-593991
22. https://www.learncpp.com/cpp-tutorial/header-files/#comment-593878
23. https://www.learncpp.com/cpp-tutorial/header-files/#comment-592977
24. https://www.learncpp.com/cpp-tutorial/header-files/#comment-592363
25. https://www.learncpp.com/cpp-tutorial/header-files/#comment-592186
26. https://www.learncpp.com/cpp-tutorial/header-files/#comment-592211
27. https://www.learncpp.com/cpp-tutorial/header-files/#comment-591238
28. https://www.learncpp.com/cpp-tutorial/header-files/#comment-590102
29. https://www.learncpp.com/cpp-tutorial/header-files/#comment-589913
30. https://www.learncpp.com/cpp-tutorial/header-files/#comment-589953
31. https://www.learncpp.com/cpp-tutorial/header-files/#comment-588988
32. https://www.learncpp.com/cpp-tutorial/header-files/#comment-588866
33. https://www.learncpp.com/cpp-tutorial/header-files/#comment-588925
34. https://www.learncpp.com/cpp-tutorial/header-files/#comment-587999
35. https://www.learncpp.com/cpp-tutorial/header-files/#comment-587689
36. https://www.learncpp.com/cpp-tutorial/header-files/#comment-587809
37. https://www.learncpp.com/cpp-tutorial/header-files/#comment-587532
38. https://www.learncpp.com/cpp-tutorial/header-files/#comment-587624
39. https://www.learncpp.com/cpp-tutorial/header-files/#comment-587530
40. https://www.learncpp.com/cpp-tutorial/header-files/#comment-587344
41. https://www.learncpp.com/cpp-tutorial/header-files/#comment-587468
42. https://www.learncpp.com/cpp-tutorial/header-files/#comment-586525
43. https://www.learncpp.com/cpp-tutorial/header-files/#comment-585351

44. https://www.learncpp.com/cpp-tutorial/header-files/#comment-584352
45. https://www.learncpp.com/cpp-tutorial/header-files/#comment-584277
46. https://www.learncpp.com/cpp-tutorial/header-files/#comment-584070
47. https://www.learncpp.com/cpp-tutorial/header-files/#comment-583624
48. https://www.learncpp.com/cpp-tutorial/header-files/#comment-583726
49. https://www.learncpp.com/cpp-tutorial/header-files/#comment-583326
50. https://www.learncpp.com/cpp-tutorial/header-files/#comment-583072
51. https://www.learncpp.com/cpp-tutorial/header-files/#comment-582934
52. https://www.learncpp.com/cpp-tutorial/header-files/#comment-583114
53. https://www.learncpp.com/cpp-tutorial/header-files/#comment-583537
54. https://www.learncpp.com/cpp-tutorial/header-files/#comment-582473
55. https://www.learncpp.com/cpp-tutorial/header-files/#comment-582331
56. https://www.learncpp.com/cpp-tutorial/header-files/#comment-582077
57. https://www.learncpp.com/cpp-tutorial/header-files/#comment-582169
58. https://www.learncpp.com/cpp-tutorial/header-files/#comment-581611
59. https://www.learncpp.com/cpp-tutorial/header-files/#comment-581887
60. https://www.learncpp.com/cpp-tutorial/header-files/#comment-581945
61. https://www.learncpp.com/cpp-tutorial/header-files/#comment-582121
62. https://www.learncpp.com/cpp-tutorial/header-files/#comment-582229
63. https://www.learncpp.com/cpp-tutorial/header-files/#comment-580825
64. https://www.learncpp.com/cpp-tutorial/header-files/#comment-580982
65. https://www.learncpp.com/cpp-tutorial/header-files/#comment-581266
66. https://www.learncpp.com/cpp-tutorial/header-files/#comment-581350
67. https://www.learncpp.com/cpp-tutorial/header-files/#comment-581348
68. https://www.learncpp.com/cpp-tutorial/header-files/#comment-580694
69. https://www.learncpp.com/cpp-tutorial/header-files/#comment-580699